Future Generation Computer Systems 28 (2012) 128–135

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Adaptive heterogeneous language support within a cloud runtime

Kathleen Ericson*, Shrideep Pallickara

Department of Computer Science, Colorado State University, Fort Collins, CO 80523, United States

ARTICLE INFO

Article history:
Received 30 December 2010
Received in revised form
10 May 2011
Accepted 28 May 2011
Available online 6 June 2011

Keywords:
Adaptive behavior
Cloud runtime
Granules
Language bridges

ABSTRACT

Cloud runtimes are an effective method of distributing computations, but can force developers to use the runtime's native language for all computations. We have extended the Granules cloud runtime with a bridge framework that allows computations to be written in C, C++, C#, Python, and R. We have additionally developed a diagnostics system which is capable of gathering information on system state, as well as modifying the underlying bridge framework in response to system load. Given the dynamic nature of Granules computations, which can be characterized as long-running with intermittent CPU bursts that allow a state to build up during successive rounds of execution, these bridges need to be bidirectional and the underlying communication mechanisms decoupled, robust and configurable. Granules bridges handle a number of different programming languages and support multiple methods of communication such as named pipes, unnamed pipes, and sockets. This choice of underlying communication mechanisms allows limited resources, such as sockets, to remain available for use by the runtime.

1. Introduction

Cloud runtimes with their support for orchestrating computations (some of which are based on the MapReduce framework [1]) have gained significant traction in the past few years. Though the cloud runtime may be developed in a specific programming language, the need often arises to orchestrate computations that have been developed in other programming languages. Here we describe our support for heterogeneous languages within Granules.

Granules [2,3] is a Java-based lightweight runtime for cloud computing and is designed to schedule a large number of computations across a set of available machines. Granules has support for both MapReduce and dataflow graphs [4]. Granules computations change state depending on the availability of data on any of their input datasets or as a result of external triggers. When the processing is complete, computations can become dormant, waiting for further data to process. This allows Granules to move away from the run-once semantics of frameworks such as Hadoop [5].

In Granules, computations specify a scheduling strategy, which govern their lifetimes. Computations scheduling strategies are defined across three dimensions: number of iterations, data availability, or periodicity. The number of iterations limits the maximum amount of times a computation can be executed. The data availability axis schedules computations as data becomes available on any input streams the computation has registered to listen to. The user can also specify that a computation should be executed on a specific interval. It is also possible to specify a custom scheduling strategy that is a combination along these three dimensions: e.g. A computation should execute no more than 1000 times, as data is available, or every 500 ms. A computation can change its scheduling strategy during execution, and Granules will enforce the newly established scheduling strategy during the next round of execution.

Computations in Granules can build state over successive rounds of execution. Though the typical CPU burst time for computations during a given execution is short (seconds to a few minutes), these computations may be long-running with computations toggling between periods of activity and dormancy. Domains that Granules is being deployed in include earthquake science, epidemiological simulations, and brain–computer interfaces [6].

While the Granules Bridge framework has been developed to work within Granules, we have found our design flexible enough to work in a basic Java environment as well as with Hadoop with millisecond overheads.

Here we describe our framework to incorporate support for computations developed in diverse programming languages within Granules. There are three important challenges that we address.

Challenge 1: *Semantics of communications between bridged computations should be independent of the mechanism to implement them.*

Bridges can be implemented in different ways. Computations in different languages should be able to use named pipes, unnamed pipes, sockets, or shared memory for communications with each other. We must abstract the content from the channel used to

* Corresponding author.
E-mail addresses: ericson@cs.colostate.edu (K. Ericson), shrideep@cs.colostate.edu (S. Pallickara).

0167-739X/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2011.05.012

implement these bridges. This allows us to introduce additional channels without having to recode the semantics of the data exchanged over the channel. An additional requirement here is for these semantics of communications to be lightweight and bidirectional while introducing acceptable overheads. The overheads introduced should not preclude the possibility for developing real-time applications in other languages.

Challenge 2: *Account for resource usage at individual machines*.

Granules is designed to support data driven computations. Computations are scheduled for execution when data is available on one of their input streams, and held dormant otherwise. Although the CPU bound processing time for individual packets of a data stream may be on the order of milliseconds, the computations are long running in the sense they are scheduled for multiple rounds of execution when incoming data packets are generated over a prolonged duration. For example, one of our benchmarks involves a Brain Computer Interface (BCI) application where the user's EEG data streams would be produced continually. Since all computations are not active at all times, Granules is capable of interleaving a large number (up to 10,000) of such computations concurrently on the same resource to maximize resource utilizations while processing streams.

Over time the availability of system resources and the accompanying performance overheads associated with using them can change. Bridges to other languages need to account for such changes. For example, (1) when a large number of processes use disk I/O for communications contentions will result in reduced response times, (2) if the number of sockets being used increases substantially configured OS thresholds would be breached resulting in errors, and (3) if shared memory is being used exclusively for communications it would result in reduced memory for applications that need them too. Since system conditions change dynamically, the framework must respond to these changes autonomously to ensure sustained system throughput.

Challenge 3: *Support reusability*.

Once a bridge to a language has been developed, this bridge functionality should be accessible to all computations written in that language. The framework needs to be reusable without requiring rewrites. Existence of a bridge to a language should imply that development of computations in that language should be just as simple as developing those in the runtime's native language. In object oriented terms, the bridge should be a base class that implements all functionality expected of computations in the native language; this base class would then be extended as needed for specific computations developed in that language.

1.1. Paper contributions

The Granules Bridge framework provides a mechanism for developers to bridge computations developed in diverse languages. Computations use the bridge to transfer information about state transitions, input datasets, results of the processing, and any errors/exceptions that occurred during the processing. We support incorporation of different communication mechanisms across bridges. Currently, our bridges to C, C++, C#, and Python can communicate via pipes. C, C++, C#, Python and R can additionally communicate using TCP; support for datagram sockets and shared memory is ongoing. This paper makes the following contributions:

Broad applicability: Though this framework was developed for a specific runtime, there is nothing here that would preclude its applicability in systems that share the need to incorporate support for other languages. For example, we have incorporated support for this framework within Hadoop, and our measured overheads here are similar to what we see in Granules. Finally, Granules is open-source and this framework has been released as part of the runtime.

Suitability to data driven computations: By dynamically adapting the communication mechanisms to changing system conditions over a period of time, computations may use a different mechanism during different rounds of execution.

Support for multiple languages: We have incorporated support for different bridging mechanisms to languages such as C, C++, C#, Python and R. This allows the runtime to orchestrate computations developed in different languages. This could also be used to create bridges that span multiple languages: for example, one may use Java as an intermediary for communications between R and Python. We have not yet benchmarked the costs for doing so.

Responsiveness to varying system conditions: The framework is lightweight and relies on diagnostics to autonomously tune communication mechanisms based on specified directives.

1.2. Organization

In Section 2, we describe the related work in this area. In Section 3 we provide details about the Granules Bridge framework as well as the Diagnostics and Adaptive Systems. We describe our experiments and report on our benchmarks in Section 4. Finally, we provide our conclusions and discuss future work in Section 5.

2. Related work

Purpose-specific wrappers are the general approach to solving communication problems between languages on a machine. While this means that wrappers can be written specifically for a particular application, and can take advantage of this lack of generality, it also means that this code is not generally reusable, and any such code can easily become difficult to maintain when extending the original program.

The Java Native Interface (JNI) [7] is a framework that allows Java programs to interact with machine-specific languages and programs from inside the JVM (Java Virtual Machine). It is designed so that C/C++ or assembly programs and Java programs can interact on a single machine. Downsides to this approach include (1) the introduction of instability to the JVM, (2) a loss of portability of Java code, and (3) the learning curve necessary to create stable code in JNI. This framework is also limited in that it only supports C/C++ and assembly programs—there is no support for other languages such as Python or C#.

Closely related to JNI is Java Native Access (JNA) (https://jna.dev.java.net/). JNA allows a developer to access system level code written in C, Windows dlls, as well as Jython [8] and JRuby (http://jruby.org/). While JNA claims to have a simpler interface, it has been reported to run approximately 100 times slower than equivalent JNI code.

CORBA [9] has been developed to handle communication across different programming languages. While it has been primarily designed to work across a network, it is possible to use it for intra-machine communication as well. A downside of using CORBA, however, is the need to create stubs and skeletons which are traversed during all communications. Additionally, communications are no longer lightweight with all the information needed to appropriately run a command.

Additionally, MPI (Message Passing Interface) has been designed to handle communications between programs executing in parallel on a single machine. MPI uses shared memory for communications, and has a high learning curve if a developer wishes to take advantage of all the functionality offered. While this does allow programs written in different languages to communicate on a single machine, it follows a different paradigm than we generally see in distributed computations.

An alternative approach for communications is through the use of XML [10]. XML allows for the development of a complex, extensible and self-descriptive language for communication, and would